Research Report 206

A Transformational Approach to Specifying Recovery in Asynchronous Communicating Systems

Zhiming Liu, Mathai Joseph

RR206

This paper describes how the transformational framework developed in [Liu91, LJ92] is applied to backward error-recovery in asynchronous communicating systems. A physical fault is modelled as an atomic action which performs state transformations in the same way as any other program action. The possible effects of a set of faults on the execution of a program are described by a transformation of the program into its fault-affected version. Fault-tolerance is provided by using transformations to add recovery actions to a non-fault-tolerant program, so that the fault-affected version of the transformed program will then satisfy a required specification. Refinement transformations can be used in the development of a fault-tolerant program. This paper provides a feasible and formal way to consider existing backward recovery techniques in terms of simple transformations.

Keywords: transformations, faults, checkpoints, recovery propagation.

Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom

January 1992

A Transformational Approach to Specifying Recovery in Asynchronous Communicating Systems*

Zhiming Liu and Mathai Joseph
University of Warwick†

Abstract: This paper describes how the transformational framework developed in [Liu91, LJ92] is applied to backward error-recovery in asynchronous communicating systems. A physical fault is modelled as an atomic action which performs state transformations in the same way as any other program action. The possible effects of a set of faults on the execution of a program are described by a transformation of the program into its fault-affected version. Fault-tolerance is provided by using transformations to add recovery actions to a non-fault-tolerant program, so that the fault-affected version of the transformed program will then satisfy a required specification. Refinement transformations can be used in the development of a fault-tolerant program. This paper provides a feasible and formal way to consider existing backward recovery techniques in terms of simple transformations.

Keywords: transformations, faults, checkpoints, recovery propagation.

1 Introduction

In [LJ92, Liu91], we described a transformational framework for developing *fault-tolerant programs*. In this framework, faults[1] are modelled as *atomic actions* which perform state transformations in the same way as other program actions (or operations). This makes it possible to extend the semantic model to include fault actions and to use a single consistent method to reason about program and fault actions. The fault actions interfere with the execution of a program in a way that is defined by the *failure semantics* of the program, derived from the semantics of the program and the fault actions. The behaviour of a program P on a system with a specified set of fault actions F_P is then simulated by a transformation of the program into a *fault-affected version* $\mathcal{F}(P, F_P)$. The semantics of such a fault-affected version of a program is then proved to be the same as the failure semantics of the program P.

Based on the failure semantics, transformations are applied to a *non-fault-tolerant* program specification. The final step of the transformations is to produce a program which is executable, efficient and fault-tolerant on a system with a specified set of faults. The transformations used in such a development procedure include *refinement transformations* [AL88, Bac87, Bac88, Bac89], *fault-tolerant transformations* and *fault-tolerant refinement transformations* [LJ92, Liu91]. Refinement transformations are required to preserve the functional correctness of non-fault-tolerant

*This work was supported in part by research grant GR/D11521 of the Science and Engineering Research Council.

†Full Address: Department of Computer Science, University of Warwick, Coventry, CV4 7AL, UK.

[1]Faults are assumed to occur only in the hardware and the execution environment, and do not include design faults in the program being developed.

programs, and allow a high level program specification to be implemented by a lower level program specification. A fault-tolerant transformation $\mathcal{R}$ provide fault-tolerance to a non-fault-tolerant program P and transforms it into $\mathcal{R}(P)$. And fault-tolerant refinement transformations are required to preserve both the functional and fault-tolerant properties of a fault-tolerant program. Such a transformational approach provides a way of converting fault properties and fault-tolerant properties into functional properties which are then amenable to the same kind of reasoning as used for the functional properties of non-fault-tolerant programs, e.g. [CM88, Lam90].

In this paper, we use transformations to introduce backward recovery into asynchronous communicating systems. There are many algorithms for recovery in such systems (e.g.[Had82, Rus80, TS84, Woo81]) and here we provide a method for their incorporation into a formal treatment of recovery in asynchronous communicating systems. To make the execution of a program P recoverable from a specified set of faults F_P, a backward recovery transformation $\mathcal{R}$ is required to transform P into a program

$$\mathcal{R}(P) \equiv P[]C_P[]P_R$$

by adding checkpointing actions C_P and recovery actions P_R. The fault-tolerant property of $\mathcal{R}(P)$ is obtained from reasoning about the program $\mathcal{F}(\mathcal{R}(P), F_P)$

The rest of this paper is organized as follows. Section 2 defines a simple model for describing and reasoning about programs, including asynchronous communicating systems. The refinement relation between programs is defined in this model. Faults and their effects are defined in Section 3. Section 4 presents a specification of the checkpointing program. Based on the definition of the consistency of a set of checkpoints (Section 5), recovery propagation is defined in Section 6. This section also shows how to find a consistent set of checkpoints. An abstract recovery program is then constructed and the correctness of the implementation of this program is reasoned about using refinement. At the end of the section, we show how to deal with failures during checkpointing and recovery. Some conclusions are drawn in Section 7 and include an assessment of the advantages of the transformational approach, a discussion about the assumptions used and comparison of this work with related work.

2 A Computational Model

As usual, we describe a program in terms of its *variables*, *states* and *actions* [Lam87]. Program variables are associated with a set of values, called the *value space*. A *state* is a mapping from the set of variables to the value space.

A *predicate* Q is a function from the set of states to the boolean values $\{true, false\}$. Thus, a state S *satisfies* Q (or S is of property Q) if $Q(S)$ is *true*.

An *action* A is a relation on the set of states. In other words, an action is a set of pairs of states. For a pair (S, S') of states, $(S, S') \in A$ means that executing A when in state S can produce (or terminate in) state S'. An action thus defines a set of transitions from state to state.

An action A is said to be *enabled* in a state S if there is a state S' such that $(S, S') \in A$. Therefore, there is a predicate gA which defines the set of states in which A is enabled. Stated in another way, $gA(S)$ is *true* if A is enabled in S. gA is called the *guard* of A.

2.1 Programs

A program P is a finite set Act_P of actions and a finite set $Init_P$ of predicates which define the *initial states* of P; the conjunction of the predicates of $Init_P$ is assumed not to be *false*. Such a program is denoted as $(Init_P, Act_P)$.

The actions of a program P are executed atomically: if an action is chosen for execution, it will be executed without interference from the other actions in the program. An *execution* of P consists of an infinite sequence, $S_0, S_1, \ldots$, of states, where S_0 satisfies all the predicates in $Init_P$ and either each pair (S_i, S_{i+1}) is in some action of Act_P, or $S_i = S_{i+1}$. A *fixed-point* of P is a state S such that for any action $A \in Act_P$,

$$\forall S' : (S, S') \in A \Rightarrow S' = S$$

Let $Fixedpoint(P)$ be a predicate defining all the fixed points of P. An execution of P, $E = S_0, S_1, \ldots$, *terminates* if it reaches a fixed point. E is said to be *fair* for an action A if for any $i > 0$, either there is a $j \geq i$ such that $(S_j, S_{j+1}) \in A$, or there is a $k \geq i$ such that $gA(S_k)$ is *false*, i.e. A cannot be enabled forever without being executed. We use $\mathcal{E}(P)$ to denote the set of executions which are fair for every action of P. Thus $\mathcal{E}(P)$ permits finite *stuttering*, i.e. in any execution, a state can be repeated consecutively at most a finite number of times before the execution terminates. This stuttering property is required for defining the refinement of programs [AL88, Bac89].

A program $P = (Init_P, Act_P)$ of n actions[2] can also be written as

$$P = (Init_P, A_1[] \ldots []A_n)$$

where $A_1, \ldots, A_n$ are actions in Act_P. If P_1 and P_2 are programs, the *union composition* program $P_1[]P_2$ is defined as

$$P_1[]P_2 \triangleq (Init_{P_2} \cup Init_{P_2}, Act_{P_1} \cup Act_{P_2})$$

where the conjunction of the predicates in $Init_{P_2} \cup Init_{P_2}$ must not be *false*.

In general, an action of a program is composed of a set of *primitive actions*. If A and B are actions, so are the *union* $A \cup B$ and the *composition* $A \circ B$, where

$$A \circ B \triangleq \{(S, S') \mid \exists S_1 : ((S, S_1) \in A) \wedge ((S_1, S') \in B)\}$$

The union operator models *choice* while the composition of actions models *sequential compositions* in programs. During the transition from one state to another, defined by an atomic action of P, the system may pass through a number of *intermediate* states such as S_1 in the definition above.

2.2 Reasoning About Programs

The Hoare triple $\{Q\}A\{R\}$ can be written as

$$\forall (S, S') : (((S, S') \in A \wedge Q(S)) \Rightarrow R(S'))$$

[2] If $Init_P$ is empty, we treat it as $\{true\}$; if Act_P is empty, we treat it as **{skip}** which is the identity (unit) relation on states.

Thus, $\{Q\}A\{R\}$ defines the total correctness[3] of action A: when A is executed in a state satisfying Q, it terminates in a state satisfying R. To reason about a program P, A is universally or existentially quantified over the actions of P; a property which holds for all points of he execution of P is defined using universal quantification while a property which holds eventually is defined using existential quantification [Lam87, Lam90, CM88]. For instance, a safety property is defined by an *invariant* Q so that Q holds initially, i.e. $Init_P \Rightarrow Q$, and $\{Q\}A\{Q\}$ holds for any action of P. For simplicity, we shall use UNITY-logic [CM88] for reasoning about programs, but we shall informally define all rules with their use.

2.3 Refinement

Refinement is a relation which defines when one program is an implementation of another. A program P is *refined* (or *implemented*) by program P', denoted $P \sqsubseteq P'$, if $\mathcal{E}(P') \subseteq \mathcal{E}(P)$. And P is *equivalent* to P', denoted by $P \equiv P'$, if they refine each other. Verification of the refinement relation of two programs can be formalized [AL88, Bac87, Bac88, Bac89]. We shall make use of these methods in our framework.

2.4 Asynchronous Communicating Systems

An *asynchronous communicating system* is a program $P = p_1[] \ldots []p_k$ consisting of k processes, each of which is a program. Assume that each pair of processes p and q share a variable ch_{pq} (called the *channel* from p to q) which is of type of sequence[4] and is used for p to send messages to q. The *sending* process p appends its message to ch_{pq}; the *receiving* process q removes the message at the head of ch_{pq}. Let cs_{pq} be the sequence of messages actually sent by p to q and let cr_{pq} be the sequence of the messages actually received from p by process q.

For each process p, let $var(p)$ be the set of *process variables* made of the following subsets:

1. X_p: the set of all the *local* variables of p,
2. $From_p \triangleq \{cs_{pq} \mid q \neq p\}$: the set of sequence variables of messages sent from p to other processes,
3. $To_p \triangleq \{cr_{qp} \mid q \neq p\}$: the set of sequence variables of messages received by p from other processes.

Each pair of processes p and q only share the variables $Ch_{pq} = \{ch_{pq}, ch_{qp}\}$. These shared variables and the variables $var(p)$ are required to satisfy the following specification.

1. The sequence of messages sent by p to q never gets shorter, i.e. for any constant sequence cs_0.

$$\textbf{stable } cs_0 \preceq cs_{pq}$$

[3]Unlike the original proposal in [Hoa69] which used the notation $Q\{A\}R$ for partial correctness.

[4]Notation: $< a, \ldots, b >$ is the sequence of elements $a, \ldots, b$; $<>$ is the empty sequence; $\sigma \wedge m$ denotes appending the element m to the end of a sequence σ; $\sigma' \preceq \sigma$ if σ' is a prefix of σ; $\sigma' \subseteq \sigma$ if each element of σ' is an element of σ but σ' preserves the order of σ; if $\sigma' \preceq \sigma$, then $\sigma - \sigma'$ is the sequence obtained from σ by removing the prefix σ'; $\#\sigma$ is the length of σ and $\sigma(i-1)$ is the ith element of σ, $1 \leq i \leq \#\sigma$; if σ is not empty, $head(\sigma)$ is the first element of σ and $rest(\sigma)$ is the sequence obtained from σ by removing $head(\sigma)$.

That is whenever $cs_0 \preceq cs_0$ becomes *true*, it will remain *true* (i.e. **stable**) for the rest of the execution.

2. A message is sent before it is received and messages are received in the order in which they were sent.

   invariant $cr_{pq} \preceq cs_{pq}$

   That is $cr_{pq} \preceq cs_{pq}$ holds initially and is **stable.**

3. Messages are not lost.

   invariant $ch_{pq} = cs_{pq} - cr_{pq}$

4. A message sent by p to q must be eventually received by q.

   $(ch_{pq} \neq <>) \longmapsto (cr_{pq} = cr_{pq} \wedge head(ch_{pq}))$

   This progress property says that if there is a message which has been sent and not received, it will eventually be received.

We use a simple *Producer-Consumer* system to illustrate the model.

Example 1 (Producer-Consumer)
A process *Producer* generates the integers $0, \ldots, n$ and sends each of them in order to a process *Consumer*. *Producer* sends the integers to *Consumer* through a channel *ch*. We can describe each process using Back's action system notation[Bac87].

process *Producer*
  initially
    $x = 0, ch =< 0 >$
  actions
    $x \leq n \rightarrow x := x + 1;\ ch := ch \wedge x$
End $\{Producer\}$

process *Consumer*
  initially
    $cr =<>$
  actions
    $ch \neq <> \rightarrow\ cr := cr \wedge head(ch);\ ch := rest(ch)$
End $\{Consumer\}$

Thus, the system is

$$Producer\text{-}Consumer \triangleq Producer[]Consumer$$

and it has the fixed point

$$x = n \wedge cr =< 0, \ldots, n >$$

3 Faults And Their Effects

Let $P = p_1[] \ldots []p_k$ be a program satisfying a specification Sp. A physical fault during the execution of a process p of P is modelled as an atomic action which transforms a *good* state into an *error* state of p, denoted by setting a boolean variable f_p to be *true*. The physical faults of p are then described by a program F_p which interferes with the execution of p. For instance, arbitrary malicious faults may set the variables of p to arbitrary values; a crash in the processor executing p may cause the variables of p to become *unavailable*; or a fault may cause the loss of a message from the channel. Such faults are represented by actions.

Assume that a fault may interfere at any point in the execution of p. If the interference by F_p on the execution of p is simply defined by the union composition $p[]F_p$, it would imply that faults occur only before or after, but not during, the execution of an atomic action of p. However, this is clearly a limited view since in practice faults do occur in intermediate states and can lead to failures.

Let each action of p be constructed using the union and composition of actions from a set of primitive actions. Interference by F_p on the execution of p can be defined as a transformation $\mathcal{F}$ in the following way.

1. For a primitive action A, $\mathcal{F}(A, F_p) \triangleq A \cup (\bigcup Act_{F_p})$
2. $\mathcal{F}(A \cup B, F_p) \triangleq \mathcal{F}(A, F_p) \cup \mathcal{F}(B, F_p)$
3. $\mathcal{F}(A \circ B, F_p) \triangleq \mathcal{F}(A, F_p) \circ \mathcal{F}(B, F_p)$
4. $\mathcal{F}(p, F_p) \triangleq (Init_p \cup \{f_p = false\}, \{\mathcal{F}(A, F_p) \mid A \in Act_p\})$

From the algebra of relations, it is easy to prove that there is a program F'_p which is derived from p and F_p such that

$$\mathcal{F}(p, F_p) \equiv p[]F'_p$$

Let F_P be $F_{p_1}[] \ldots []F_{p_k}$. The *fault-affected version* of P is the program

$$\mathcal{F}(P, F_P) \triangleq \mathcal{F}(p_1, F_{p_1})[] \ldots []\mathcal{F}(p_k, F_{p_k})$$

Note that the program $\mathcal{F}(P, F_P)$ may not satisfy the original specification Sp.

Different failure behaviours of a program, under different assumptions of faults, can be described as refinements of the corresponding fault-affected versions of the program. For example, assume that $\mathcal{F}(P, F_1)$, $\mathcal{F}(P, F_2)$ and $\mathcal{F}(P, F_3)$ exhibit respectively malicious failure, processor crash and fail-stop failure. Then typically

$$\mathcal{F}(P, F_1) \sqsubseteq \mathcal{F}(P, F_2) \sqsubseteq \mathcal{F}(P, F_3)$$

4 Checkpointing

Backward recovery usually take place from *checkpoints* at which the local state of a process is recorded for later restoration. However, the restoration of the state of one process must be

accompanied by restoring the program as a whole to a *consistent* state (informally, a consistent state is one that could have occurred during the execution of the program), and this may require recovery from checkpoints of other processes in the program. In this section we shall specify a means of checkpointing that is inherently nondeterministic, with no restriction on *when* or *where* a process takes its checkpoints. In the next section, we shall describe how recovery is made to a consistent state.

Given a program P with a set $Proc$ of processes, a checkpoint of any process p contains the values of the process variables in $var(p)$. Let rec_x be a sequence variable which records the saved values of variable x. At a checkpoint, let the checkpointing program append the value of each process variable x to rec_x.

Let $Rec_p = \{rec_x \mid x \in var(p)\}$ be the set of all the recorded variables of p. A checkpointing state of process p is a state over Rec_p, denoted as CP_p. Rec_p is a set of sequence variables and at a checkpoint the values of all the variables in $var(p)$ are required to be simultaneously appended to the corresponding variables in Rec_p. Thus, all the variables in Rec_p have the same length $\#CP_p$ in each state.

The *checkpointing program* for program P is required to satisfy the specification:

1. for any $rec_x, rec_y \in Rec_p$,

   invariant $\#rec_x = \#rec_y$

2. for process $p \in Proc$ and any $x \in var(p)$, the recorded values of x are previous values of x,

   invariant $rec_x \subseteq ob(x)$

   where $ob(x)$ is an auxiliary (history) variable recording the history of values of x.

3. no checkpoint can be taken when a fault occurs in process p and a fault cannot modify variables used for checkpointing, i.e. for any constant sequence c_0,

   stable $f_p \wedge rec_x = c_0$

For a process $p \in Proc$, define the action $Save_p$ by a multiple assignment with a guard $\neg f_p$ (so that the action will only take place when there is no failure in p)

$$Save_p = \neg f_p \rightarrow \|_x : x \in var(p) :: rec_x := rec_x \wedge x$$

and the initial predicates

$$Init'_p \triangleq \{rec_x =< x > \mid x \in var(p)\}$$

We can then write a simple checkpointing program C_P for the program P with no restriction on when or where checkpoints will be taken. If P has the processes $p_1, \ldots, p_k$, the checkpointing program C_P is

$$C_P = (\bigcup_{p \in Proc} Init'_p, Save_{p_1} [] \ldots [] Save_{p_k})$$

For any process p_i let $C_{p_i} \triangleq (\{rec_x =< x > \mid x \in var(p_i)\}, Save_{p_i})$ and $p'_i \triangleq p_i [] C_{p_i}$. Then $P [] C_P$ is refined by

$$P' \triangleq p'_1 [] \ldots p'_k$$

5 Consistent Checkpoints

A *local state* of a process p is a state S_p over $var(p)$. A global state of a program P, described in Section 2, can be partitioned into the local states of its processes and the state of the channels. But the state of channels is determined by the local states of the processes: during the execution of the communicating processes of P, the invariants 2 and 3 of Section 2.4. must always be true. These are then conditions for the consistency of local states.

Given a positive number m such that $m \leq \#CP_p$, the function $CP_p(m)$ is defined as follows: for each variable $x \in var(p)$

$$CP_p(m)(x) = CP_p(rec_x)(m)$$

where $CP_p(m)$ is a local state of p, called the mth checkpoint of process p.

For two processes p and q, $p \neq q$, and checkpoints $CP_p(m)$ and $CP_q(n)$ of p and q, the predicate $SemiCon(CP_p(m), CP_q(n))$ (i.e. *semi-consistent*) is defined as

$$SemiCon(CP_p(m), CP_q(n)) \triangleq CP_p(m)(cr_{pq}) \propto CP_q(n)(cs_{pq})$$

The *consistency* of $CP_p(m)$ and $CP_q(n)$, denoted $Consistent(\{CP_p(m), CP_q(n)\})$ is then defined to be

$$SemiCon(CP_p(m), CP_q(n)) \wedge SemiCon(CP_q(n), CP_p(m))$$

If process p should have to restart from a checkpoint $CP_p(m)$ because of the occurrence of a fault, process q will also have to restart from a checkpoint $CP_q(j)$ if the current state $\widehat{q}$ of q is not consistent with $CP_p(m)$. But when a fault occurs, it may not be necessary for all the processes of P to recover from an earlier point. Let $\mathcal{P} \subseteq Proc$ and $\overleftarrow{\mathcal{P}}$ be a set of checkpoints of processes in $\mathcal{P}$. For each process $p \in \mathcal{P}$, there is exactly one checkpoint $C_p(i_p)$ of p in $\overleftarrow{\mathcal{P}}$. Then $\overleftarrow{\mathcal{P}}$ is a *recovery line* of the process variables in $\mathcal{P}$ if

1. $Consistent(\overleftarrow{\mathcal{P}})$
2. $\forall p \in \mathcal{P},\ \forall q \notin \mathcal{P} : Consistent(CP_p(i_p), \widehat{q})$

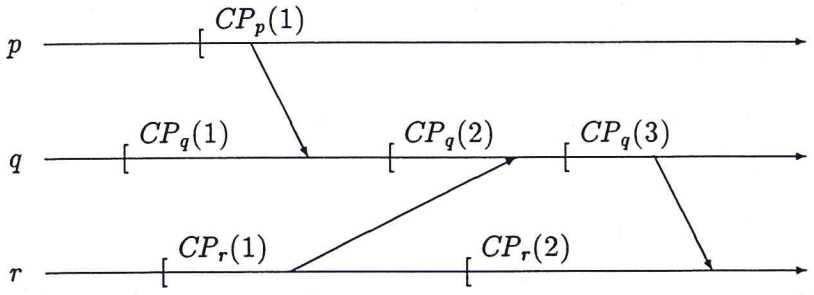
The next section shows how to find a recovery line.

6 Recovery Transformation

6.1 Recovery Propagation

This section presents a way for determining a backward recovery line for a set of failed processes. For this, we shall define a linear order $\prec$ on the checkpoints of a process.

- A checkpoint $CP_p(m)$ of p is *earlier than* a checkpoint $CP_p(i)$ of p, denoted by $CP_p(m) \prec CP_p(i)$, if $m < i$; $CP_p(i)$ is also said to be *later than* $CP_p(m)$.
- A checkpoint $CP_p(m)$ is *not later than* a checkpoint $CP_p(i)$, denoted $CP_p(m) \preceq CP_p(i)$, if $CP_p(m)$ is earlier than $CP_p(i)$ or $m = i$.



$CP_p(1) \rightsquigarrow CP_q(1) \rightsquigarrow CP_r(2) \quad CP_p(1) \overset{*}{\rightsquigarrow} CP_r(2)$

Figure 1: Direct and indirect propagator relations

- Given a subset $\mathcal{P}$ of *Proc*, let $\mathcal{CP}$ and $\mathcal{CP}'$ be sets of checkpoints such that for each process $p \in \mathcal{P}$, there is exactly one checkpoint $CP_p(i_p)$ of process p in $\mathcal{CP}$ and exactly one checkpoint $CP_p(j_p)$ of process p in $\mathcal{CP}'$. Then,

  $$\mathcal{CP} \preceq \mathcal{CP}' \triangleq \forall\, p \in \mathcal{P} : CP_p(i_p) \preceq CP_p(j_p)$$

  and

  $$\mathcal{CP} \prec \mathcal{CP}' \triangleq (\mathcal{CP} \preceq \mathcal{CP}') \wedge \exists\, q \in \mathcal{P} : (CP_q(i_q) \prec CP_q(j_q))$$

Consider two processes p and q in *Proc* such that $p \neq q$. If process p should have to restart from a checkpoint $CP_p(m)$ because of the occurrence of a fault, process q will have to restart from a checkpoint, say $CP_q(j)$, if the current state $\widehat{q}$ of q is not consistent with $CP_p(m)$.

$CP_p(m)$ is said to be a *recovery direct propagator* for $CP_q(j)$, denoted $CP_p(m) \rightsquigarrow CP_q(j)$, if it satisfies the following conditions:

1. $SemiCon(CP_p(m), CP_q(j))$
2. $\neg \exists\, CP_q(j') : (CP_q(j) \prec CP_q(j')) \wedge SemiCon(CP_p(m), CP_q(j'))$
3. $\neg SemiCon(CP_p(m), \widehat{q})$

The *indirect propagator* relation $\overset{*}{\rightsquigarrow}$ between checkpoints is defined to be the reflexive and transitive closure of the direct propagator $\rightsquigarrow$ relation:

1. $CP_p(m) \overset{*}{\rightsquigarrow} CP_p(m)$,
2. $CP_p(m) \rightsquigarrow CP_q(j) \Rightarrow CP_p(m) \overset{*}{\rightsquigarrow} CP_q(j)$,
3. $(CP_p(m) \overset{*}{\rightsquigarrow} CP_t(i) \overset{*}{\rightsquigarrow} CP_q(j)) \Rightarrow CP_p(m) \overset{*}{\rightsquigarrow} CP_q(j)$.

The direct and indirect propagator relations model the *domino-effect* described in [Ran75], and are illustrated in Figure 1, in which an arrow from one process to another represents the despatch or receipt of a message.

To obtain the lemmas, theorems and corollaries in this section, checkpointing and recovery are required to satisfy the following assumption[5].

Assumption 1 *If $CP_p(m)$ and $CP_p(i)$ are checkpoints of a process $p \in Proc$, then for any process $q \in Proc$ such that $q \neq p$, then*

1. $CP_p(m)(cs_{pq}) \cong (\widehat{cs}_{pq}) \wedge CP_p(m)(cr_{qp}) \cong \widehat{cr}_{qp}$
2. $(CP_p(m) \preceq CP_p(i)) \Rightarrow CP_p(m)(cs_{pq}) \cong CP_p(i)(cs_{pq}) \wedge CP_p(m)(cr_{qp}) \cong CP_p(i)(cr_{qp})$

The following results can then be proved (see [Liu91] for proofs).

Theorem 1 *A set of checkpoints $\overleftarrow{\mathcal{P}}$ is a recovery line of $\mathcal{P}$, iff*

1. $\forall p \in \mathcal{P} : \exists! CP_p(m) \in \overleftarrow{\mathcal{P}}$ *(where $\exists!$ means there is exactly one)*
2. $\forall CP_p(i), CP_q(j) : ((CP_p(i) \in \overleftarrow{\mathcal{P}}) \wedge (CP_p(i) \overset{*}{\rightsquigarrow} CP_q(j)) \Rightarrow q \in \mathcal{P})$
3. *Consistent*$(\overleftarrow{\mathcal{P}})$

In Figure 1, the following sets of checkpoints are recovery lines:

$\{CP_p(1), CP_q(1), CP_r(1)\}$ is a recovery line of $\{p, q, r\}$;

$\{CP_p(1), CP_q(1), CP_r(2)\}$ is a recovery line of $\{p, q, r\}$;

$\{CP_q(2), CP_r(1)\}$ is a recovery line of $\{q, r\}$;

$\{CP_q(2), CP_r(2)\}$ is a recovery line of $\{q, r\}$;

$\{CP_q(3), CP_r(2)\}$ is a recovery line of $\{q, r\}$.

But there are no other recovery lines in this figure.

The set of checkpoints propagated by $CP_p(m)$ is the *recovery domain* of $CP_p(m)$ and is defined as

$$Z(CP_p(m)) \triangleq \{CP_q(i) \mid q \in Proc \wedge CP_p(m) \overset{*}{\rightsquigarrow} CP_q(i)\}$$

Lemma 1 *Given processes p and q and checkpoints $CP_p(i)$, $CP_p(j)$ and $CP_q(m)$ such that $CP_p(i)$ is not later than $CP_p(j)$,*

$$(CP_p(j) \rightsquigarrow CP_q(m)) \Rightarrow \exists\, CP_q(l) : (CP_q(l) \preceq CP_q(m)) \wedge CP_p(i) \rightsquigarrow CP_q(l)$$

This lemma means that if by recovering at a checkpoint $CP_p(j)$, process p causes process q to recover from a checkpoint $CP_q(m)$, then by recovering at a checkpoint which is not later than $CP_p(j)$, p causes q to recover from a checkpoint which is not later than $CP_q(m)$.

[5]This assumption can be implemented in the following way: when process p recovers from a fault using checkpoint $CP_p(m)$, the checkpoints of process p that have been established after $CP_p(m)$ must be deleted, i.e. the checkpointing program also has to recover. The assumption is reasonable because if process p recovers from $CP_p(m)$, the states of p later than $CP_p(m)$ are unusable (see Section 6.2).

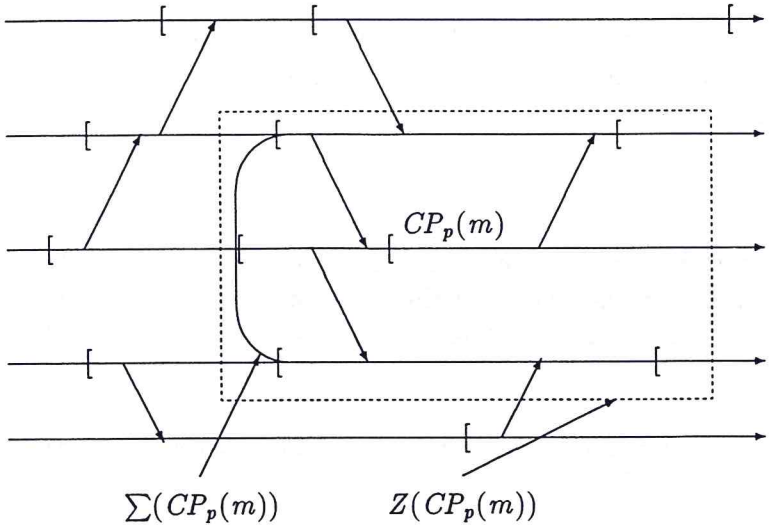


Figure 2: Illustration of Lemma 2

Lemma 2 *The set of checkpoints*

$$\sum(CP_p(m)) \triangleq \{CP_q(i) \mid CP_q(i) \in Z(CP_p(m)) \wedge \neg\exists\, CP_q(i') \in Z(CP_p(m)) : CP_q(i') \prec CP_q(i)\}$$

is the recovery line of the set of processes

$$RL(CP_p(m)) \triangleq \{q \mid \exists\, CP_q(i) \in \sum(CP_p(m))\}$$

in which recovery propagation may appear.

Therefore, the earliest checkpoints in the recovery domain of checkpoint $CP_p(m)$ represent a recovery line with respect to $RL(CP_p(m))$. This is illustrated in Figure 2.

The following theorems characterize some important properties of checkpointing and recovery.

Theorem 2 *For each process $p \in Proc$ and checkpoints $CP_p(m)$ and $CP_p(i)$ of p,*

$$(CP_p(m) \preceq CP_p(i)) \Rightarrow RL(CP_p(i)) \subseteq RL(CP_p(m))$$

This says that in rolling back to an earlier checkpoint, a process may cause additional processes also to roll back.

Theorem 3 *The recovery line defined by Lemma 2 is the* **latest** *recovery line for $RL(CP_p(m))$ w.r.t. $CP_p(m)$, i.e. if $\mathcal{CP}$ is a set of consistent checkpoints such that for each process q in $RL(CP_p(m))$ there is one and only one checkpoint $CP_q(n_q) \in \mathcal{CP}$ and $CP_p(n_p) \preceq CP_p(m)$, then*

$$\mathcal{CP} \preceq \sum(CP_p(m))$$

Lemma 3 *For the set of checkpoints $\sum(CP_p(m))$ defined in Lemma 2, for each process $q \in RL(CP_p(m))$,*

$$\exists\, CP_r(i) \in \sum(CP_p(m)) : \neg SemiCon(CP_r(i), \widehat{q})$$

Theorem 4 *The set of recovery processes $RL(CP_p(m))$ determined in Lemma 2 is the* **smallest** *set of processes that have to recover w.r.t. $CP_p(m)$, i.e. for any recovery line $\overleftarrow{\mathcal{P}}$ of $\mathcal{P} \subseteq Proc$ that contains a checkpoint $CP_p(i)$ of process p such that $CP_p(i) \preceq CP_p(m)$,*

$$RL(CP_p(m)) \subseteq \mathcal{P}$$

When a program recovers after the detection of errors, the recovery line should be contained in the recovery domain of the latest checkpoint of the failed process. This suggests that it would be useful to perform the *smallest* and *latest* backward recovery. Let $\widehat{rec}_p$ denote the *latest* checkpoint of p.

Corollary 1 *For each process $p \in Proc$, $\sum(\widehat{rec}_p)$ is the* **smallest** *and* **latest** *recovery line w.r.t. process p: i.e. for each recovery line $\overleftarrow{\mathcal{P}}$ such that $p \in \mathcal{P}$,*

$$RL(\widehat{\text{rec}}_p) \subseteq \mathcal{P}$$

and,

$$\forall\, q \in RL(\widehat{\text{rec}}_p) \cap \mathcal{P} :: (CP_q(m) \in \textstyle\sum(\widehat{\text{rec}}_p) \wedge CP_q(j) \in \overleftarrow{\mathcal{P}} \Rightarrow (CP_q(j) \preceq CP_q(m)))$$

If faults may occur simultaneously in different processes, the recovery actions can be coordinated by *concatenating* the recovery lines. The concatenation of two recovery lines is defined as follows:

$$\begin{aligned}&\overleftarrow{\mathcal{P}}_1 \uplus \overleftarrow{\mathcal{P}}_2 \triangleq \\ &\quad \{CP_p(m) \mid CP_p(m) \in ((\overleftarrow{\mathcal{P}}_1 \cup \overleftarrow{\mathcal{P}}_2) - (\overleftarrow{\mathcal{P}}_1 \cap \overleftarrow{\mathcal{P}}_2)) \\ &\vee (\exists\, CP_p(m_1) \in \overleftarrow{\mathcal{P}}_1, CP_p(m_2) \in \overleftarrow{\mathcal{P}}_2 : (m = min\{m_1, m_2\}))\}\end{aligned}$$

The concatenation of two recovery lines $\overleftarrow{\mathcal{P}}_1$ and $\overleftarrow{\mathcal{P}}_2$ is a recovery line of the set $\mathcal{P}_1 \cup \mathcal{P}_2$.

Corollary 2 *The concatenation $\sum(CP_p(m)) \uplus \sum(CP_q(j))$ for two checkpoints $CP_p(m)$ and $CP_q(j)$ is the latest and smallest recovery line w.r.t. $CP_p(m)$ and $CP_q(j)$: i.e. any recovery line $\overleftarrow{\mathcal{P}}$ which contains $CP_p(m)$ and $CP_q(j)$ satisfies the following conditions*

1. $\overleftarrow{\mathcal{P}} \preceq \sum(CP_p(m)) \uplus \sum(CP_q(j))$,
2. $\mathcal{P} \subseteq (RL(CP_p(m)) \cup RL(CP_q(j)))$.

6.2 Recovery Program

From this discussion, we can conclude that a recovery program P_R for program P behaves in the following way. After a fault occurs in a process $p \in Proc$, a recovery line containing the process p is calculated using the recovery propagation relation. The processes in this recovery line are then restored to checkpoints corresponding to the recovery line. And the channels of these processes are restored to the states appropriate for those checkpoints (note that this may require that all outgoing messages from a process are stored between checkpoints).

We present below a general recovery program P_R for a program P but first we shall define some notation. Let $\mathcal{P}$ be a subset of $Proc$ and

$$\mathcal{CP} = \{CP_p(m_p) \mid p \in \mathcal{P}\}$$

be a set of checkpoints such that for each process $p \in \mathcal{P}$ there is exactly one checkpoint $CP_p(m_p)$ of p in $\mathcal{CP}$. $\mathcal{CP}(x)$ is used to denote the value $CP_p(m_p)(x)$ for $x \in var_p$. For each pair of processes $p \in \mathcal{P}$ and $q \in Proc$ such that $p \neq q$,

1. $ch_{pq}.rec(\mathcal{CP}) \triangleq CP_p(m_p)(cs_{pq}) - CP_q(m_q)(cr_{pq})$, if $q \in \mathcal{P}$,
2. $ch_{pq}.rec(\mathcal{CP}) \triangleq CP_p(m_p)(cs_{pq}) - \widehat{cr}_{pq}$, if $q \notin \mathcal{P}$,
3. $ch_{qp}.rec(\mathcal{CP}) \triangleq \widehat{cs}_{qp} - CP_p(m_p)(cr_{qp})$, if $q \notin \mathcal{P}$.

$ch_{pq}.rec(\mathcal{CP})$ and $ch_{qp}.rec(\mathcal{CP})$ can be simplified to $v.ch_{pq}$ and $v.ch_{qp}$ respectively when $\mathcal{CP}$ is given explicitly.

From Corollary 1, the smallest and latest recovery line $\sum(\mathcal{P})$ for the set $R(\mathcal{P})$ of processes can be defined as:

$$\sum(\mathcal{P}) \triangleq \biguplus_{p \in \mathcal{P}} \sum(\widehat{rec}_p)$$

and

$$R(\mathcal{P}) \triangleq \bigcup_{p \in \mathcal{P}} RL(\widehat{rec}_p)$$

For the set $\mathcal{P}$ of processes, let

$$f_{\mathcal{P}} \triangleq \forall p \in Proc : (p \in \mathcal{P} \Leftrightarrow f_p)$$

Program P_R:
$[]_{\mathcal{P}} : \mathcal{P} \subseteq Proc :: f_{\mathcal{P}} \rightarrow (\|_x : x \in Var(P) :: x := v_x).REC_P$
End $\{P_R\}$

The general recovery program has an action which is enabled when faults occur in any processes; when it is enabled, the command assigns to the variables of P the values satisfying REC_P. The predicate REC_P defines a recovery line $\mathcal{CP}$ for the set $RL(\mathcal{CP})$ of processes as the conjunction of the following predicates:

1. $\mathcal{P} \subseteq RL(\mathcal{CP})$;
2. $\forall p \in RL(\mathcal{CP}) : x \in var(p) \Rightarrow v_x = \mathcal{CP}(x)$;
3. $\forall p \in RL(\mathcal{CP}) \forall q \in Proc : v_{ch_{pq}} = v.ch_{pq}$;
4. $\forall q \in RL(\mathcal{CP}) \forall p \in Proc : v_{ch_{pq}} = v.ch_{pq}$;
5. $\forall p \notin \mathcal{P} : x \in var(p) \Rightarrow v_x = x$;
6. $\forall p, q \notin \mathcal{P} : v_{ch_{pq}} = ch_{pq}$;

7. $\forall p \in Proc : v_{f_p} = false.$

Let

$$\mathcal{R}(P) \triangleq P[]C_P[]P_R$$

The fault-tolerant properties of $\mathcal{R}(P)$ are obtained from the properties of the program

$$\mathcal{F}(\mathcal{R}(P), F_P)$$

If it can be assumed that no faults occur during the execution of the checkpointing program C_P and the recovery program P_R, we have

$$\mathcal{F}(\mathcal{R}(P), F_P) \equiv \mathcal{F}(P, F_P)[]C_P[]P_R$$

In implementations of P_R, only variables in the recovery line need to be reassigned, and not all the variables of P. We can prove the correctness of a fault-tolerant implementation P' of P by proving that $\mathcal{F}(P', F_P)$ refines $\mathcal{F}(\mathcal{R}(P), F_P)$. If we choose $\mathcal{CP}$ as $\sum(\mathcal{P})$ and $RL(CP)$ as $R(\mathcal{P})$, P_R is refined to the latest and smallest recovery program $P_{R_{ls}}$. From Corollary 1, a recovery program which performs the smallest and latest recovery behaves as follows. Should a fault occur in a process $p \in Proc$, the recovery program simultaneously restores the processes in the recovery line to the checkpoints corresponding to the recovery line *w.r.t.* the latest checkpoint $\widehat{rec}_p$ of process p. It also restores the corresponding channels of the processes in $RL(\widehat{rec}_p)$ to states determined by those checkpoints. When faults occur in more than two processes and cause these processes to recover simultaneously, the restoration is done by the concatenation of the recovery lines *w.r.t.* the latest checkpoints of these processes.

6.3 Failure During Recovery

In the previous two sections, for simplicity, we assumed that no faults occur during checkpointing and recovery. This assumption makes it possible to have the equation

$$\mathcal{F}(\mathcal{R}(P), F_P) \equiv \mathcal{F}(P, F_P)[]C_P[]P_R$$

The fault-tolerant properties of $\mathcal{R}(P)$ are then obtained by reasoning about the functional properties of the program on the right handside of the equation.

If faults may occur during recovery, we can deal with them by slightly changing the recovery transformation $\mathcal{R}$ in the following way. Assume that the faults which may occur during the execution of P_R are represented by a program F_R and that f_r is *true* after such a fault occurs. We change the multiple assignment

$$(||_x : x \in Var(P) :: x = v_x).REC_P$$

in P_R into a loop

$$\textbf{do } f_r \rightarrow (||_x : x \in Var(P) :: x = v_x).REC_P \textbf{ od}$$

This does not itself need any checkpoints since the consistent checkpoints are re-calculated after a faults occurs in P_R. (This is similar to the method described in [SS83] for dealing with failure

during recovery.) In this case, the fault-tolerant properties of $\mathcal{R}(P)$ are obtained from the program

$$\mathcal{F}(P, F_P)[]C_P[]\mathcal{F}(P_R, F_R)$$

Thus the transformational approach can be extended to deal with faults during recovery.

There are other ways of dealing with this problem, e.g. by detecting the failure of the recovery process by some other process of P, or by having nested recovery actions (so that when one recovery process fails, another is automatically invoked). Transformations to achieve such recovery can be similarly constructed.

We can treat a failure during checkpointing in a similar way to that of a failure during recovery. For example, a slight change in the actions of the checkpointing program will allow an action to be repeated when a fault occurs. It is also possible to consider the corruption of checkpoints by faults. In this case, the recovery line has to be composed of uncorrupted checkpoints, and it must be assumed that there is at least one such a recovery line (e.g. the initial state of P).

7 Conclusions and Discussion

7.1 Conclusions

Using the transformational framework developed in [LJ92, Liu91], this paper presents a formal analysis of static and dynamic recovery in asynchronous communicating programs.

Dynamic checkpointing and recovery can be done in two ways. In the first approach, processes independently save checkpoints. On the occurrence of a fault, the processes must coordinate to find a consistent set of saved checkpoints from which to establish a recovery line. The system is then rolled back and restarted from this recovery line. Techniques for the provision of fault-tolerance in communicating systems using this approach, such as [Rus80, Woo81, Had82], provide a class of refined versions of $P[]C_P[]P_R$. A sub-class of these refined version is the class of the refined versions of $P[]C_P[]P_{R_{ls}}$, which represent the latest and smallest recovery algorithms.

Using the second approach, processes coordinate their checkpointing actions so that each process saves only its most recent checkpoint, and the set of of checkpoints in the system is guaranteed to be consistent. When a fault occurs, the system restarts from these checkpoints [TS84]. This can also be treated as a refined version of $P[]C_P[]P_R$.

With static checkpointing and recovery mechanisms, the recovery line is determined statically and the recording variables therefore do not need to be sequences variables. Recovery propagation can be restricted to lie within fault-tolerant structures, e.g. no recovery propogation is needed between a process inside a *conversation* and any process outside the conversation [Ran75]. Application of the transformational approach to formal treatment of this kind of recovery can be found in [Liu91].

Only backward recovery is considered in this paper. Application of the transformational framework to forward recovery methods is considered in [Liu91] which also contains the proofs of the theorems in Section 6.

7.2 About the Assumptions

For simplicity, we have assumed a built-in fairness condition in the model so that we can use UNITY-logic in this work. This fairness assumption can be removed and, instead of UNITY, a more powerful logic such as TLA [Lam90] could be used in which it is possible to express various fairness conditions.

Proving the progress properties of a fault-tolerant program, e.g. $\mathcal{F}(\mathcal{R}(P), F_P)$, requires some *finite error behaviour assumption*, e.g. that faults can only occur finitely often in each execution of the program. By placing a guard on a fault action of F_P to disable the action when the guard is *false*, various finite error behaviours can be modelled [Liu91].

The boolean variable f_p of a process is used to model the assumption that faults are detectable. There may be many different causes of failure, such as a crash of a processor, loss of messages, or a fail-stop error and their detection can be represented by different predicates and f_p treated as their disjunction. It is known that without some assumption about detectability, fault-tolerance cannot be achieved [FLP85].

We have assumed here that the programs are *closed systems* and that recovery is always possible by restoring processes to a consistent state. If a program interacts with its environment, it will by itself not be a closed system and recovery of the program must be accompanied by the restoration to a consistent state of the closed system. For example, it may be necessary that after recovery, communications with external devices not be repeated, or that the program must be restored to a possible future state (i.e. using forward recovery) that is consistent with the states of the external devices. Investigations in previous work [Liu91] show that more complex assumptions about recovery can be accommodated in the transformational framework, but with some additional (and acceptable) complexity.

7.3 Related Work

The idea that physical faults be modelled as another kind of action (or operation) was described in [Cri85] through an example. There, process crashes and faults that affect the physical storage medium were taken to be special operations, referred to *fault operations*, and described by axioms similar to those used to describe the semantics of ordinary operations. We have developed this idea into a general framework by introducing transformations, so that methods for refinement of programs can be applied to fault-tolerance. An advantage is that it allows us to treat various aspects of fault-tolerance in an uniform way.

Methods for formal treatment of fault-tolerance in distributed programs were proposed in [Schl82, Sch82, SS83, SS84]. There, specific failure assumptions are encoded in the semantic model and the corresponding proof rules for ordinary programs were changed for dealing with the assumed faliures. For example, in [Schl82, Sch82, SS83], *fail-stop* semantics were used and proof rules were proposed for dealing with *fail-stop* processes; and in [SS84], proof rules for communication primitives of networks suporting unreliable datagrams were proposed. In our framework, we have encoded a set of faults as actions and represented their effect by a transformation. We believe that this provides a more feasible way of dealing with various kind of faults in a single consistent model, including for example multiple occurrences of faults. The same methods and rules are used to reason about fault-tolerant programs as for ordinary programs. Further, refinement is also defined into this framework so that fault-tolerance can be achieved by refinement.

Acknowledgements: We would like to thank Anders Ravn and Michael Hansen for some useful discussions and for their comments, and Fred Schneider for his comments on the work in [Sch82, SS83, SS84]. Part of the work of Zhiming Liu was done during a visit to the Technical University of Denmark and was supported by the **RapID** Project.

References

[AL88] M. Abadi and L. Lamport. The existence of refinement mapping. In *Proc. 3rd IEEE Sympoium on Logic and Computer Science*, 1988.

[Bac87] R.J.R. Back. A calculus of refinement for program derivations. Technical Report 54, Abo Akademi, 1987.

[Bac88] R.J.R. Back. Refining atomicity in parallel algorithms. Technical Report 57, Abo Akademi, 1988.

[Bac89] R.J.R. Back. Refinement calculus, Part II: Parallel and reactive programs. Technical Report 93, Abo Akademi, 1989.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley Publishing Company, 1988.

[Cri85] F. Cristian. A rigorous approach to fault tolerant programming. *IEEE Transactions on Software Engineering*, SE-11(1):23–31, January 1985.

[Had82] V. Hadzilacos. An algorithm for minimizing rollback cost. In *Proceedings of ACM Symposium on Principles of Database Systems.*, March 1982.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[Lam87] L. Lamport. *win* and *sin*: Predicate transformers for concurrency. Technical Report 17, Systems Research Center of Digital Equipment Corporation in Palo Alto, California, May 1987.

[Lam90] L. Lamport. A temporal logic of actions. Technical report, Digital SRC, California, April 1990.

[Liu91] Z. Liu. *Fault-tolerant Programming by Transformations.* PhD thesis, Department of Computer Science, University of Warwick, September, 1991.

[LJ92] Zhiming Liu and Mathai Joseph. Transformation of programs for fault-tolerance. *Formal Aspects of Computing*, (to appear), 1992.

[Ran75] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[Rus80] D.L. Russell. State restoration in systems of communication processes. *IEEE Transactions on Software Engineering*, SE-6(2):183–194, March 1980.

[Schl82] R.D. Schlichting. *Axiomatic verification to enhance software reliability.* PhD thesis, Department of Computer Science, Cornell University, January 1982.

[Sch82] F.B. Schneider. Synchronization in distributed programs. *ACM Transactions on Programming Languages and Systems*, 4(2):125–148, April 1982.

[SS83] R.D. Schlichting and F.B. Schneider. Fail-stop processes: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, 1983.

[SS84] R.D. Schlichting and F.B. Schneider. Using message-passing for distributed programming. *ACM Transactions on Programming Languages and Systems*, 6(3):402–431, July 1984.

[TS84] Y. Tamir and C.H. Sequin. Error recovery in multicomputers using global checkpoints. In *Proceedings of 13th International Conference on Parallel Processing*, August 1984.

[Woo81] W.G. Wood. A decentralized recovery control protocol. In *Proceedings of the 11th Annual International Symposium on Fault-Tolerant Computing*, June 1981.